CAS Front-End Engineering

JS CLOSURES, MODULES, INHERITANCE

Silvan Gehrig

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Lernziele

Die Vorlesung soll den Teilnehmer befähigen, die Objekt-orientierten Mechanismen von JavaScript zu erkennen, diese Konzepte fachgerecht einzusetzen und entsprechende Architektur-Entscheide zu treffen.

Die Teilnehmer...

- **verstehen den Unterschied zwischen Scope und Context in Zusammenhang mit OOP.**
- **können MVC in Front-End Web Applikationen in OOP-Manier einsetzen.**
- **kennen Nested Function Scopes (Closures) und können Scope/Context-Fehler erkennen und beheben.**
- **kennen EcmaScript 2015 Module und können diese im Projekt anwenden.**

Aus Vorkurs:

- **können eigene JavaScript Klassen mit EcmaScript 2015 (ES6) kreieren.**

Table of Contents

Table of Contents

BASICS: JAVASCRIPT

TYPE SYSTEM

Primitive Types I

- **boolean** — ***true* or *false***
- **number** — ***0* or *1*, *2*, *3*, etc**
- **string** — **"*string*"**
- **(*Symbol*) ES2015**
- **undefined** — ***undefined* (writeable until ES5)**
- **(null)**

Primitive Types II

- **General Behavior of Primitive Types**
  - Immutable
  - Copy by value
  - Call by value
  - Compared by value

  - *Auto-Boxing if used as Reference Type*

Reference Types I

repetition

- ***Object*** — ***new Object() or { }***
  - Boolean — *new Boolean()*
  - Number — *new Number()*
  - String — *new String()*
  - Date — *new Date()*
  - Array — *new Array()* or *[ ]*
  - RegExp — *new RegExp()* or */reg-exp/*
  - Function — *new Function()* or *function() { }*

- null (as type, null-value is a primitive, similar to [0x000000])

Remarks:
Don't use primitive type constructors in your code!

Reference Types II

- **General Behavior of an Object**
  - Objects are similar to dictionaries
  - Every reference type inherits from Object
  - Copy by reference
  - Call by reference
  - Compared by reference

  - *Auto-Unboxing by calling .valueOf()*

STRICT MODE

Strict Mode

repetition

- **Indicates that the code should be executed in "strict mode"**
  - It's a literal expression, ignored by earlier versions of JavaScript
  - Declared at the beginning of a JavaScript file, or a JavaScript function
- ***Strict Mode* converts mistakes into errors**
  - The following condition will throw an error:
    - Assigning a
      - non-writable property
      - a getter-only property
      - a non-existing property
      - a non-existing variable
      - a non-existing object
  - Prohibits keywords (e.g. with())
  - **this can be undefined (or null), if function isn't called in an objects context**
- **EcmaScript 2015 Classes / Methods / Modules are executed in strict mode**

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

NESTED SCOPES
CLOSURES

BASICS: JAVASCRIPT

Nested Scopes

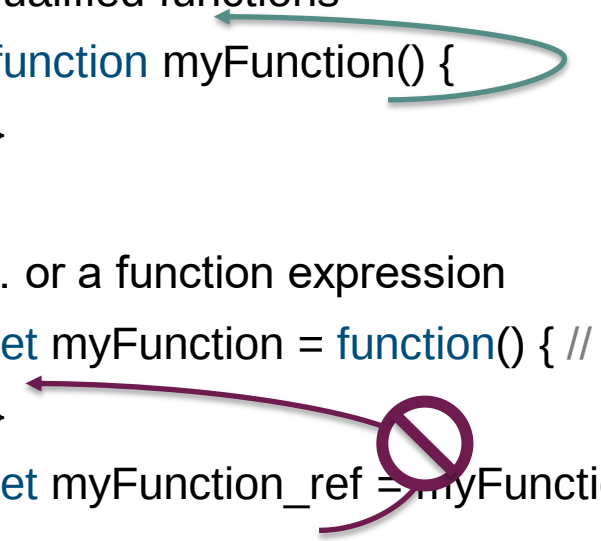
repetition

- **Function Definition**
  - Qualified functions

```
function myFunction() {
}
```

  - … or a function expression

```
let myFunction = function() { // name given by assignment
}
let myFunction_ref = myFunction;
```



Nested Scopes

- **Functions define a new Scope**

```
function myFunction() {
    let inner = 5;
}
// outer scope, no access to inner
```

- **… also allowed**

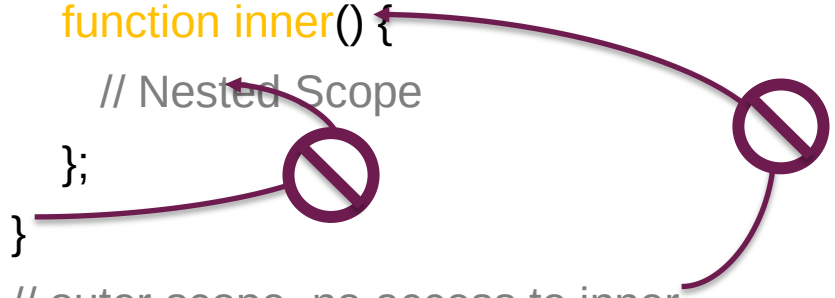
```
function myFunction() {
    let inner = function() { // declares a Nested Scope
    };
}
// outer scope, no access to inner
```

Nested Scopes

repetition

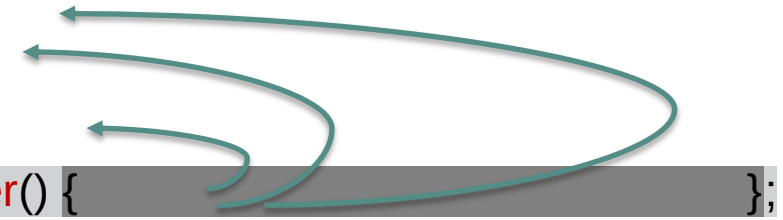
- **Nesting of qualified/named functions**

```
function myFunction() {
    function inner() {
      // Nested Scope
    };
}
// outer scope, no access to inner
```



- **… chaining nested functions**

```
function myFunction() {
    let inner = function() {
        function nestedInner() {                    };
    };
}
```



Closures

- **Accessing variables from outer scope**

```
function myFunction() {
    let inner = 5;
    let getInner = function() {
        return inner;
    };
    return getInner;
}
```

```
let getInnerPtr = myFunction(); // returns the function pointer to getInner
let innerValue = getInnerPtr(); // execute getInner function
// innerValue contains 5
```

CONTEXT

BASICS: JAVASCRIPT

Definition of an OO-Language: Everything is an Object

- ☑ **Objects communicate by sending and receiving messages**
- ☑ **Objects have their own memory**
- ☑ **Every object is an instance of a class**
- ☑ **The class holds the shared behavior for its instances**
- **To eval a program list, control is passed to the first object and the remainder is treated as its message**

Alan Kay*: *The Early History of Smalltalk (1993)

Objects have their own memory

- **Context**
  - Represents the “own” Object’s memory
  - Can be accessed by using the *this variable*
  - this references properties and methods of the current object

  - A new Object is created by using the *new operator*
    - This will switch the Context (this) to the created Object’s memory

Context Code Example in JavaScript ES5

legacy

```
function House (color) {                    // class definition, constructor function
    this.facadeColor = color;               // property definition
    this.paint = function(newColor) {       // method definition
        this.facadeColor = newColor;        // do more paint stuff here, colorize windows, etc…
    };
}

var whiteHouse = new House("white");        // whiteHouse represents an instance (House object)
whiteHouse.paint("beige");
```

“Abnormal” Context behavior ES5 I

legacy

- **Object is instantiated by using new keyword**
- **BUT a JavaScript class is also a *function***
  - Can also be called regularly without new
  - Context doesn’t change; global context is injected

```
function House (color) {                    // class definition, constructor function
    this.facadeColor = color;
    this.paint = function(newColor) {
        this.facadeColor = newColor;
    };
}
var whiteHouse = House("white");            // used without new operator
```

facadeColor property and paint() method are written into the **global context**!

“Abnormal” Context behavior ES5 II

legacy

- **A method is called by declaring the object as context**
- **BUT a JavaScript method is also a *function***
  - Can also be called regularly without the context
  - Context doesn’t change; global context is injected

```
function House (color) {                    // class definition, constructor function
    this.facadeColor = color;
    this.paint = function(newColor) {
        this.facadeColor = newColor;
    };
}
var whiteHouse = new House("white");
var paintWhiteHouse = whiteHouse.paint; // copy pointer of function paint
paintWhiteHouse();                          // call function without object (without context)
```

facadeColor property is written into the **global context**!

Context Code Example in JavaScript ES2015

```
class House {                          // class definition
    constructor(color) {               // constructor definition
        this.facadeColor = color;      // property definition
    }
    paint (newColor) {                 // method definition
        this.facadeColor = newColor;   // do more paint stuff here, colorize windows, etc…
    };
}

let whiteHouse = new House("white");   // whiteHouse represents an instance (House object)
whiteHouse.paint("beige");
```

“Abnormal” Context behavior ES2015 I

- **Object is instantiated by using `new` keyword**
- **BUT an ES2015 class is also a *function***
  - `typeof` operator returns “function”
- **`class` constructors cannot be invoked without `new`**
  - Results in a runtime error
  - More deterministic than ES5 approach

“Abnormal” Context behavior ES2015 II

- **A method is called by declaring the object as context**
- **BUT a JavaScript method is also a *function***
  - Can also be called regularly without the context
  - Context doesn’t change; ‘undefined’ is used instead (strict mode behavior)

```
class House {                                   // class definition
    constructor(color) { this.facadeColor = color; }
    paint (newColor) {
        this.facadeColor = newColor;
    };
}
let whiteHouse = new House("white");
let paintWhiteHouse = whiteHouse.paint;         // copy pointer of function paint
paintWhiteHouse();                              // call function without object (without context)
```

this is ‘undefined’, writing the facadeColor property will result in a **runtime error**!

SCOPE VS CONTEXT

BASICS: JAVASCRIPT

Scope / Context Summary

- **Scope is defined by its function chain**
  - Variables on scope are newer «lost»

- **Context is bound according the function call**
  - Context changes according the precedent (left-hand) object
  - Behavior can be used for polymorphism

Real World Example when using existing Objects

repetition

```
class House {                                   // existing class definition, given by a framework
    constructor(color) { this.facadeColor = color; }
    paintWhite() {
        this.facadeColor = "white";
    }
}

const house = new House("red");                 // don't use …addEventListener("click", house.paintWhite);
document.getElementById("BtnPaintWhite").addEventListener("click", function() {
        house.paintWhite();
} );
document.getElementById("BtnPaintWhite").addEventListener("click", house.paintWhite.bind(house) );
```

Scope / Context Conclusion

As a recommendation…

…use *Closures (or Lambdas) with scoped variables* or *bind()* if you have to use function pointers

But there are several side effects when applying Closures

- **Access to modified Closure when using `functions()`**
- **Breaks some native language features**

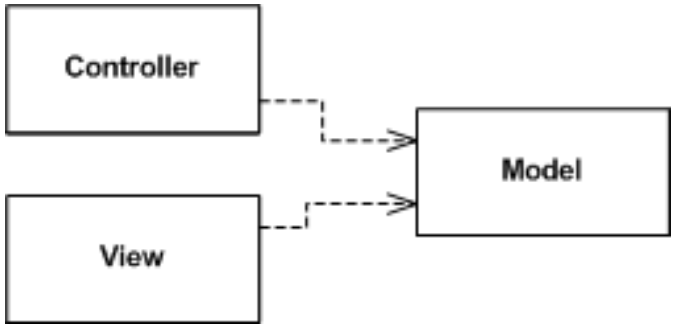
MODEL VIEW CONTROLLER

BASICS: OOP *reloaded*

Model View Controller (MVC) - Introduction

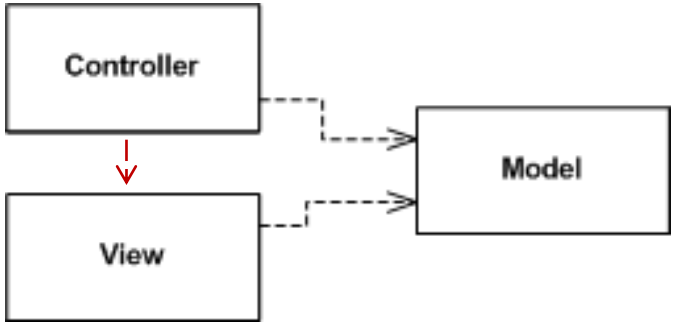
- **Design Pattern described by Martin Fowler**
  - https://www.martinfowler.com/eaaDev/uiArchs.html#ModelViewController
- **…more structured description can be found in POSA Volume 1**
  - http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471958697.html

- **One of the first attempts to do serious UI work (in the ’70ies)**
- **Heart of MVC is what Fowler calls *Separated Presentation***
- **Domain objects (logic) should work**
  - **without reference** to the presentation
  - ...and they should also be able to **support multiple presentations**



MVC - Structure and Collaborators

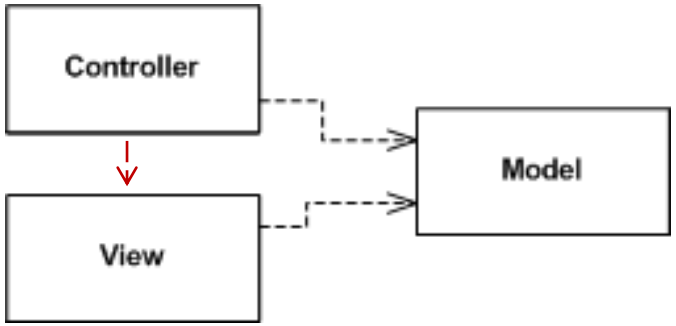
- **In MVC domain element (business logic, core functionality and data) is referred as the Model**
- **Presentation part of MVC is made of the two remaining elements**
  - The **Controller**'s job is to take the user's input
    …and figure out what to do with it
  - **View** is responsive for displaying the state of the model



MVC - Consequences

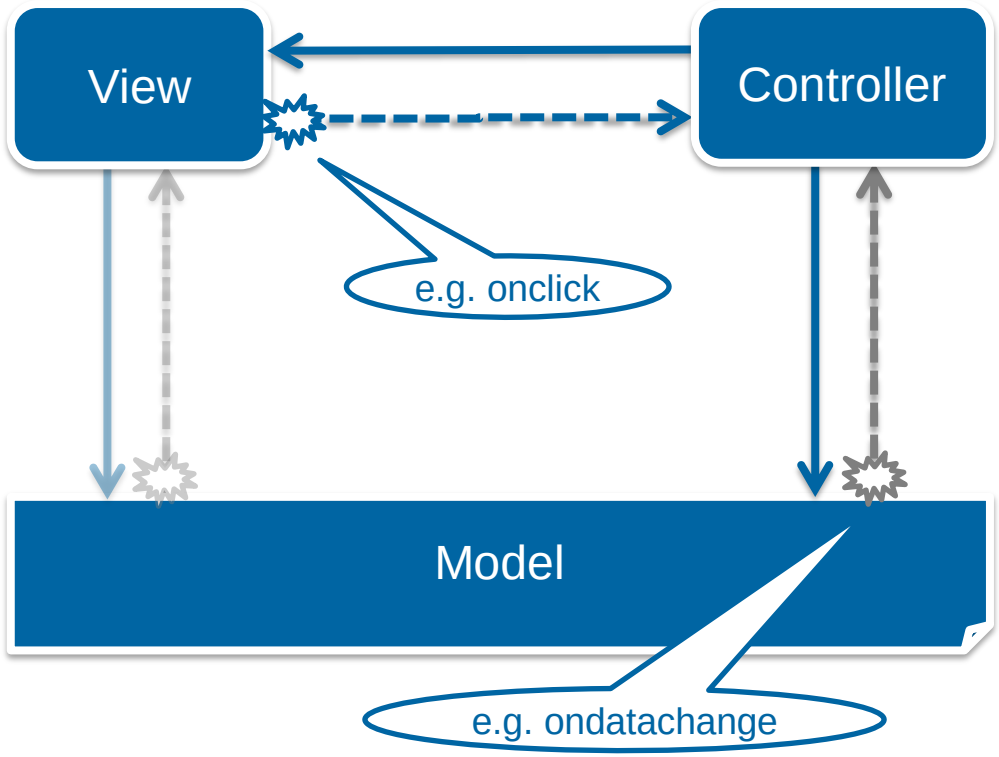
- **Benefits**
  - It decouples the **V**iew from the **M**odel (logic).
  - It abstracts how objects cooperate (**C**ontroller)
  - Multiple **V**iews can represent data of the same **M**odel

- **Liabilities**
  - It centralizes control
  - “Massive View Controller”
  - Many implementations and variations available
    - M-V-VM vs MVC vs MVP vs Component Architecture…



MVC *and the Web*

- **Model**
  - Data and business logic for the application
- **View**
  - Displays the state/data of the model
- **Controller**
  - Takes the user's input
  - Mediates between the View and the business logic
  - …uses routing features
    (more later on in Angular / React lectures)



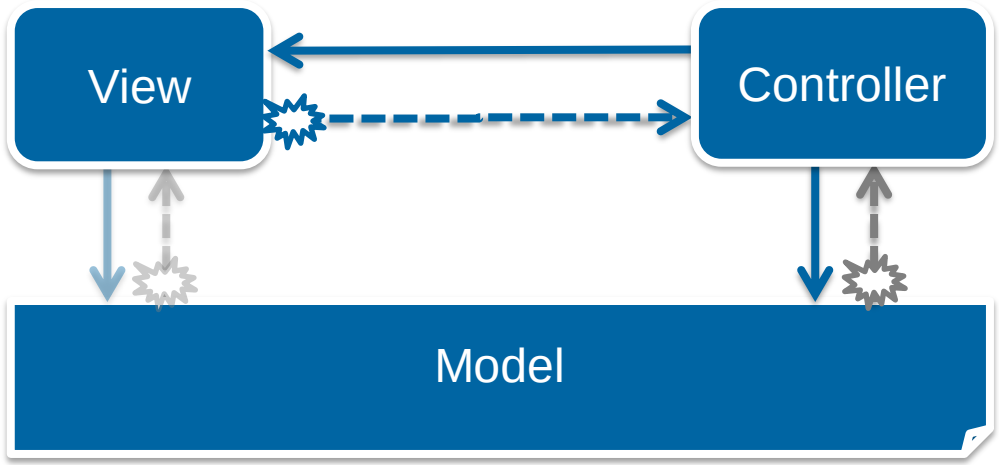
MVC *and the Web - Vanilla Example*

- **Model: `food.js`**

```
class Food {
    constructor() { }  /* … */
}
```

- **View: `zoo.html`**

```
<html>
    <body>
    …
    <!-- handlebars templates -->
    </body>
</html>
```

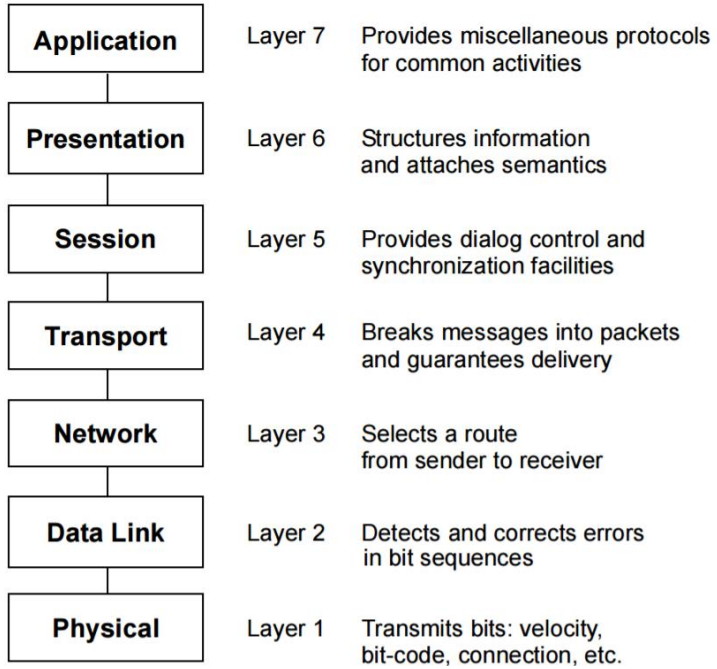


- **Controller: `zoo-controller.js`**

```
class ZooController {
    constructor(uiElement) {
            uiElement.onclick = /* … */
    }
}
```

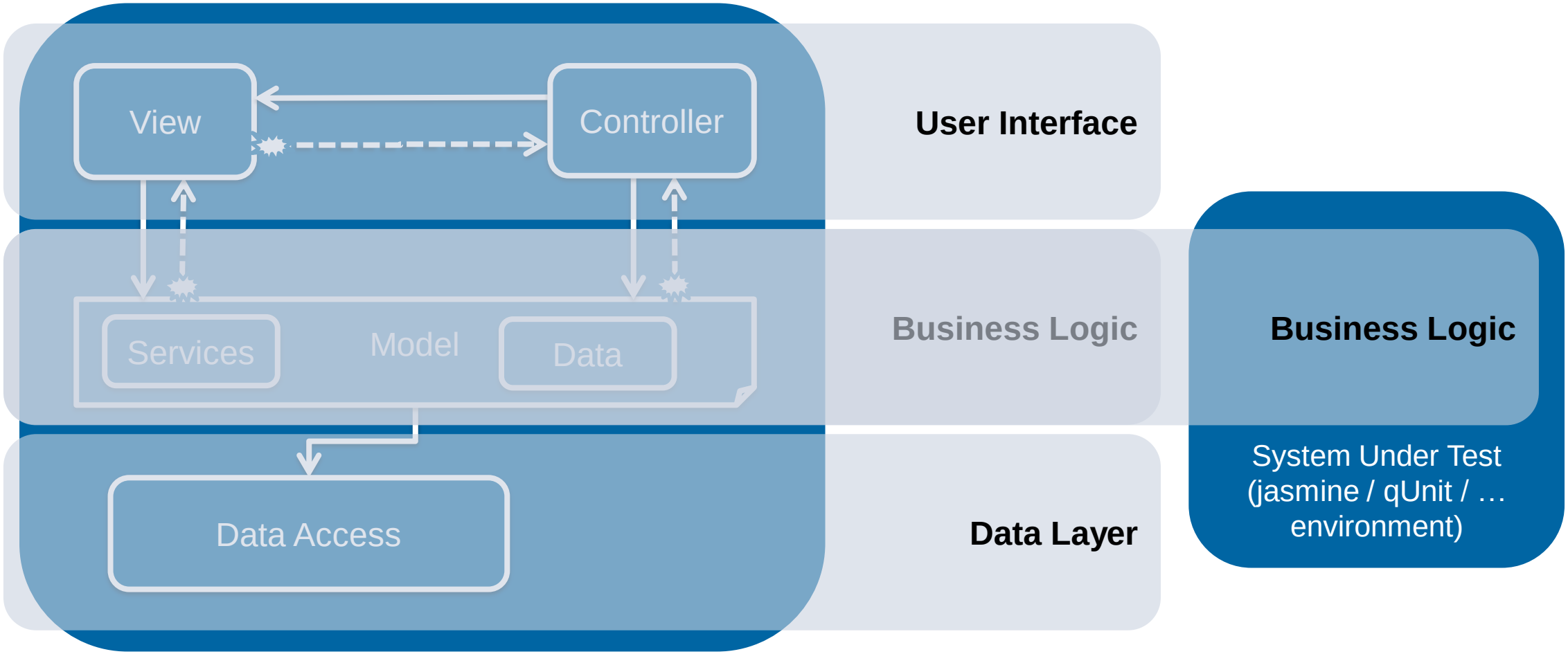
Layering Basics

- **Due its characteristics, MVC is used when client code gets more complex**
- **Layering means structuring the client scripts into composed groups of subtasks**
- **Each subtask is at a particular level of abstraction**



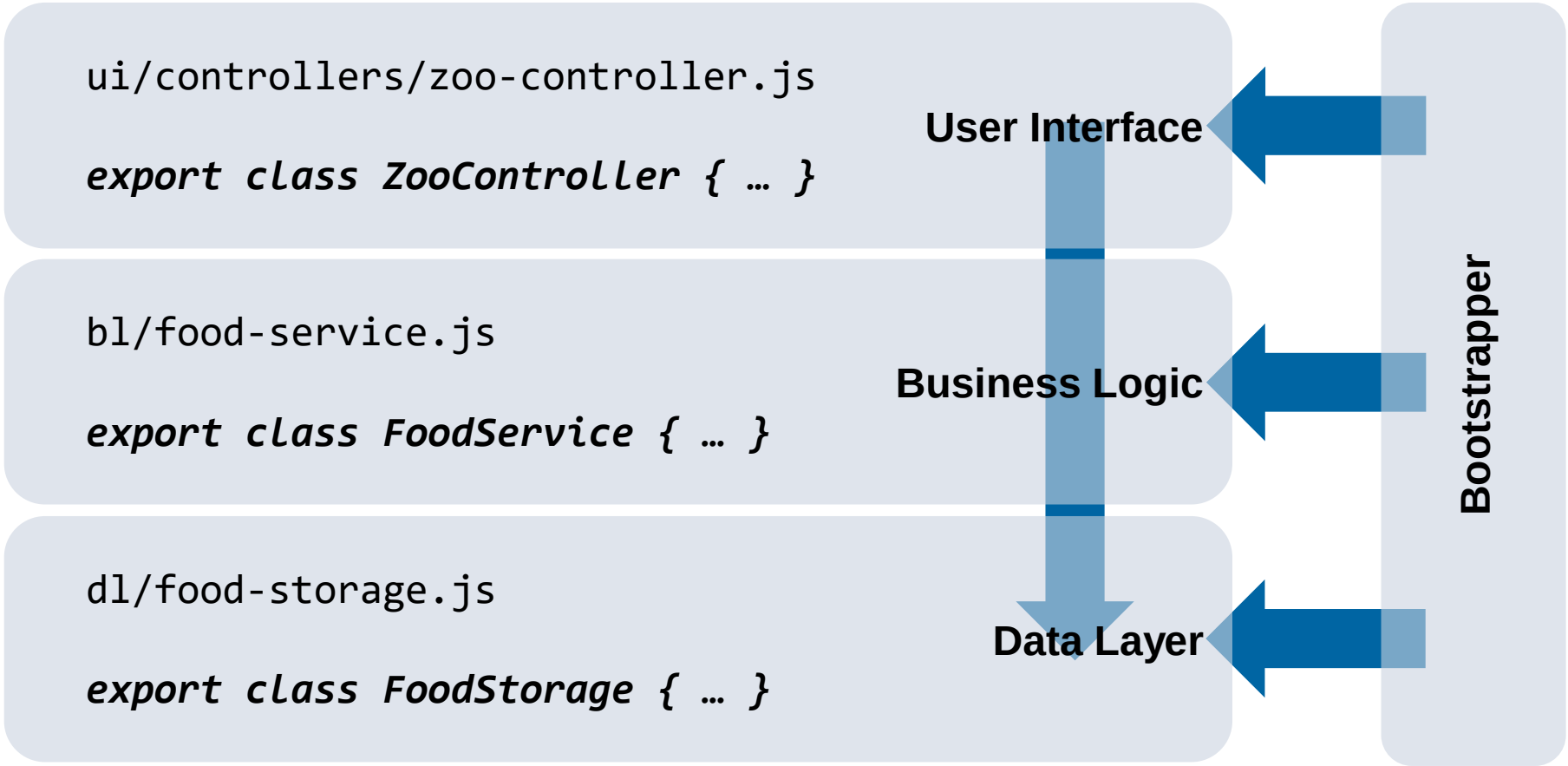
Source: PoEAA 03, Martin Fowler

MVC + S *in a Web Application*



Bootstrapping an MVC + S *Web Application*

demo



Exercise 1 – JavaScript strukturieren

30min

Clonen Sie die Aufgaben von https://github.com/IFS-Web/HSR.CAS-FEE.JS-ENG2 und öffnen Sie das index.html File.

Die Anleitung zum Aufsetzen der Übung befindet sich direkt auf dem GitHub Repository.

- **Lernziele**

  Den Bestehenden Code auf File-System Basis gemäss Layering strukturieren (logische Gliederung der JavaScript-Konstrukte).

- **Lesen Sie zuerst die Aufgabe bis zum `Exercise 1` durch.**
- **Lösen Sie `Exercise 1` (siehe Aufgaben im index.html File).**

OOP WITH JAVASCRIPT

ES2015 CLASS SYSTEM & INHERITANCE

(AUS VORKURS)

OOP WITH JS

Class Syntax

- **Class as expression**

```
const Foo = class {
    constructor() {}
    bar() {
      return "Hello World!";
    }
};
```

- **Class definition**

```
class Foo {
    constructor() {}
    bar() {
      return "Hello World!";
    }
};
```

Static Member Syntax

- **Static Method**

```
class Bar {
    constructor(value) {
        this.value = value;
        // this.classMethod not possible!
    }
    static classMethod() {
            return 'hello';
    }
};
```

- **Usage**

```
Bar.classMethod();
```

Property Syntax

- **Getter / Setter Methods**

```
class Bar {
    constructor(value) {
        this.value = value;
    }
    get aProp() {
         return 'getter';
    }
    set aProp(value) {
       console.log('setter: '+value);
    }
};
```

- **Usage**

```
let fooBar = new Bar();
let getterValue = fooBar.aProp;
fooBar.aProp = 'new-value';
```

Private Members (ES.Next / TC39 Stage 3)

- **Private Fields / Methods**

```
class Bar {
    #value = 0;
    constructor(value) {
        this.#value = value;
    }
    get prop() {
        return this.#value;
    }
    set #prop(value) {
        console.log('setter: ' + this.#value);
        this.#value = value;
    }
};
```

- **Usage**

```
let fooBar = new Bar();
let getterValue = fooBar.prop;
```

https://github.com/tc39/proposal-private-methods
https://github.com/tc39/proposal-class-fields

Inheritance Syntax

- **Base Class**

```
class Bar { // implicitly extends Object
    constructor(value) {
        this.value = value;
    }
};
```

- **Derived Class**

```
class Foo extends Bar {
    constructor(value) {
        super(value);
    }
    toString() {
        return `Foo: ${super.toString()}`;
    }
};
```

- **Super-class's constructor can be called by using super();**
  - If parent class defines a constructor, **super()** must be called

Method Overriding Syntax

- **Base Class**

```
class Bar {
    constructor() {
    }
    do() {
        // do something in base method
    }
};
```

- **Derived Class**

```
class Foo extends Bar {
    constructor() {
        super();
    }
    do() { // override base class's do() method
        super.do(value);
    }

};
```

- **Super-class's methods can be called by using super.*[methodName]*()**
  - Super-class methods/properties are still available even when overridden
- **Properties (getter/setter) can be invoked by using super.*[propertyName]***
  - But: *Instance fields* cannot be accessed by using super statement.

Advanced: Class Syntax vs Structured Programming I

- **Structured Programming**
  ES4 / ES5, single instance

```
var Calculator = {
    left: 0,
    right: 0,
    sum: function() {
      return this.left + this.right;
    }
};
```

- **Class definition**
  ES2015, single (default) instance

```
class Calculator {
    constructor() {
        this.left = 0;
        this.left = 0;
    }
    sum() {
        return this.left + this.right;
    }
}

Calculator.default = new Calculator();
```

Advanced: Class Syntax vs Structured Programming II

- **Structured Programming**
  ES4 / ES5, multiple instances

```
function createCalculator() {
    return {
        left: 0,
        right: 0,
        sum: function() {
          return this.left + this.right;
        }
    };
}
var calculator1 = createCalculator();
var calculator2 = createCalculator();
```

- **Class definition**
  multiple instances possible

```
class Calculator {
    constructor() {
        this.left = 0;
        this.left = 0;
    }
    sum() {
        return this.left + this.right;
    }
}

const calculator1 = new Calculator();
const calculator2 = new Calculator();
```

Exercise 2 – Objektorientiertes JavaScript

30min

Arbeiten Sie weiter an Ihrer Lösung aus Übung 1.

- **Lernziele**

  Strukturiertes (objektbasiertes) JavaScript in objektorientierte Klassen umbauen.

  Siehe Folien «Class Syntax vs Structured Programming»

- **Lösen Sie `Exercise 2` (siehe Aufgaben im index.html File).**
- **Im Abschnitt "Mögliche Vorgehensweise" finden Sie eine Anleitung, wie die Aufgabe gelöst werden kann.**
  - Versuchen Sie zuerst die Aufgabe selbstständig zu lösen und nehmen Sie die "Mögliche Vorgehensweise" später zurat.

PATTERNS AND IDIOMS

OOP WITH JS

Introduction

- **An Idiom**
  - …is a recurring construct bound to a programming language or technology
  - …describes a simple feature that is not built-in

- **(Design) Patterns**
  - … are more complex than Idioms
  - … are proven solutions of a specific problem
  - … can be easily reused
  - … can be expressive

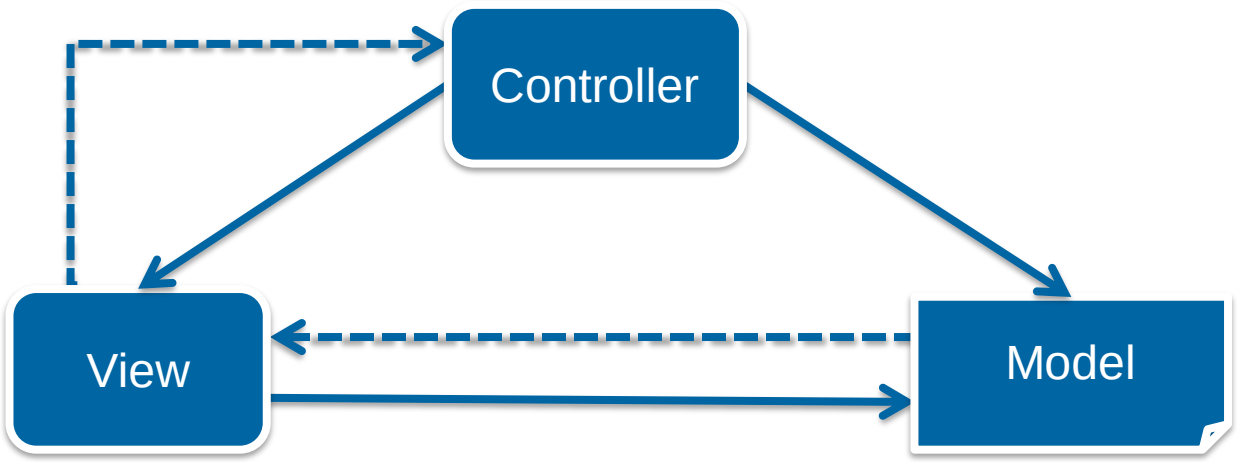
Example of an Idiom

- **Idiom “Swapping values between variables”**

```
let a = 5;
let b = 3;
function swap() {
    let temp = a;
    b = a;
    a = temp;
}
```

Example of a Pattern

- **Pattern “MVC” by Martin Fowler**



- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation

Idiom: Defensive Programming

```
paint(newColor) {                          // ES2015 method paint()
    newColor = String(newColor);           // enforce newColor as String
    // do more stuff with newColor here…
}
```

- **Intent**
  - Enforce correct type of arguments
  - Can also be combined with other defensive programming mechanisms (e.g. Precondition Checks)
- **Variations**
  - Prefix arguments directly with their primitive types

```
paint(strNewColor) {
    // do more stuff with newColor here…
}
```

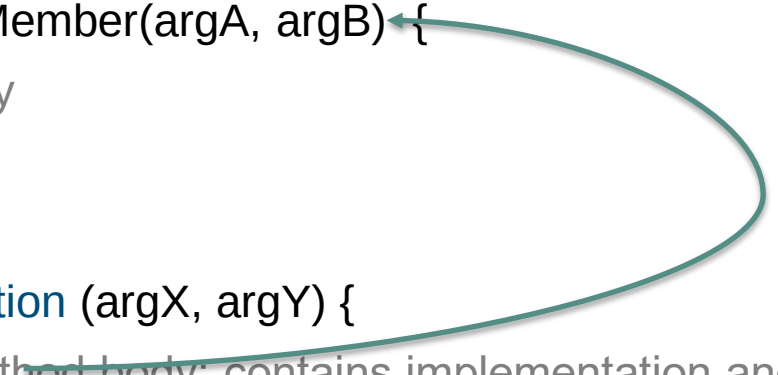
Idiom: Immediately-invoked Function Expression (IIFE)

```
;( function() {
    'use strict';
    // closure scope, do your stuff here
} ());
```

- **Intent**
  - Keep the global scope clean
  - Strict mode applied in a controlled environment

(Revealing) Module Pattern I

```
let moduleName = (function() {
    let privateVar;
    function privateMember(argA, argB) {
        // method body
    }
    return {
        member: function (argX, argY) {
            // public method body; contains implementation and uses private members…
        }
    };
}()); // use it with moduleName.member()
```



(Revealing) Module Pattern II

- **Intent**
  - Local variables are fully shielded from global scope
  - Variables existence is limited to within the module's closure
  - Modules cover namespacing, public, and private variables
- **Usages**
  - AMD modules (later)
  - common.js modules (later in node.js lecture)
  - ES2015 modules

Source: http://addyosmani.com/resources/essentialjsdesignpatterns/book/

ASYNCHORNOUS MODULES

Synchronous Modules Syntax

- **Ad hoc Modules (Module Patterns)**
  - Are represented by JavaScript Patterns and natively supported by all EcmaScript 5+ compatible runtimes
    - Module Pattern
    - Singleton Pattern
    - …
  - Normally, these modules are loaded in **synchrounous** manner by using `<script> </script>` tags

```
let m = (function() { … })();
```

Asynchronous Modules Introduction

- **Loading all modules in synchronous manner may result in a performance hit**
- **Asynchronous loading (on-demand) is preferred**
  - Especially on client-side (browser) to avoid slow startup times
  - In most cases, an additional framework (e.g. RequireJS, SystemJS) is needed
  - …or even further approach:
    - Directly bundle and minify your application in a few larger files (as kind of «module library»)
    - *...not part of this lesson, see JSPM, WebPack and other frameworks*

- **Multiple asynchronous modules syntax**
  - Asynchronous Module Definition (AMD) `define`
  - CommonJS Standard Syntax (CJS) `require / module.…`
  - ES2015 Modules (ESM) `import / export`

Comparing Asynchronous Modules Syntax

- **CommonJS Standard Syntax (CJS)** `require / module.…`
  - CommonJS Syntax (CJS) is used in conjunction with **node.js modules**
  - Natively supported in node.js
  - Requires a framework, such as require.js when using in a web browser
- **Asynchronous Module Definition (AMD)** `define`
  - AMD Syntax is mostly used for websites
  - Also requires a framework (e.g. require.js) when using in a web browser
- **EcmaScript 2015 Modules (ESM)** `import / export`
  - Modules with asynchronous import and export syntax are part of the EcmaScript 2015 (ESM)
  - Still limited browser/runtime support

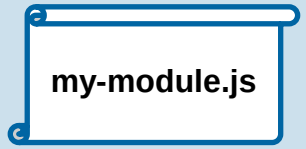
Asynchronous Module Definition (AMD) I

```
// calling define with dependency array, and factory function (which will return the module)
define('my-module', ['dep1', 'dep2'], function (dep1, dep2) {

    // now we can use module ‘dep1’ in argument dep1, ‘dep2’ in argument dep2

    // module pattern implementation

    return {      // return public variables & functions
    };
});
```



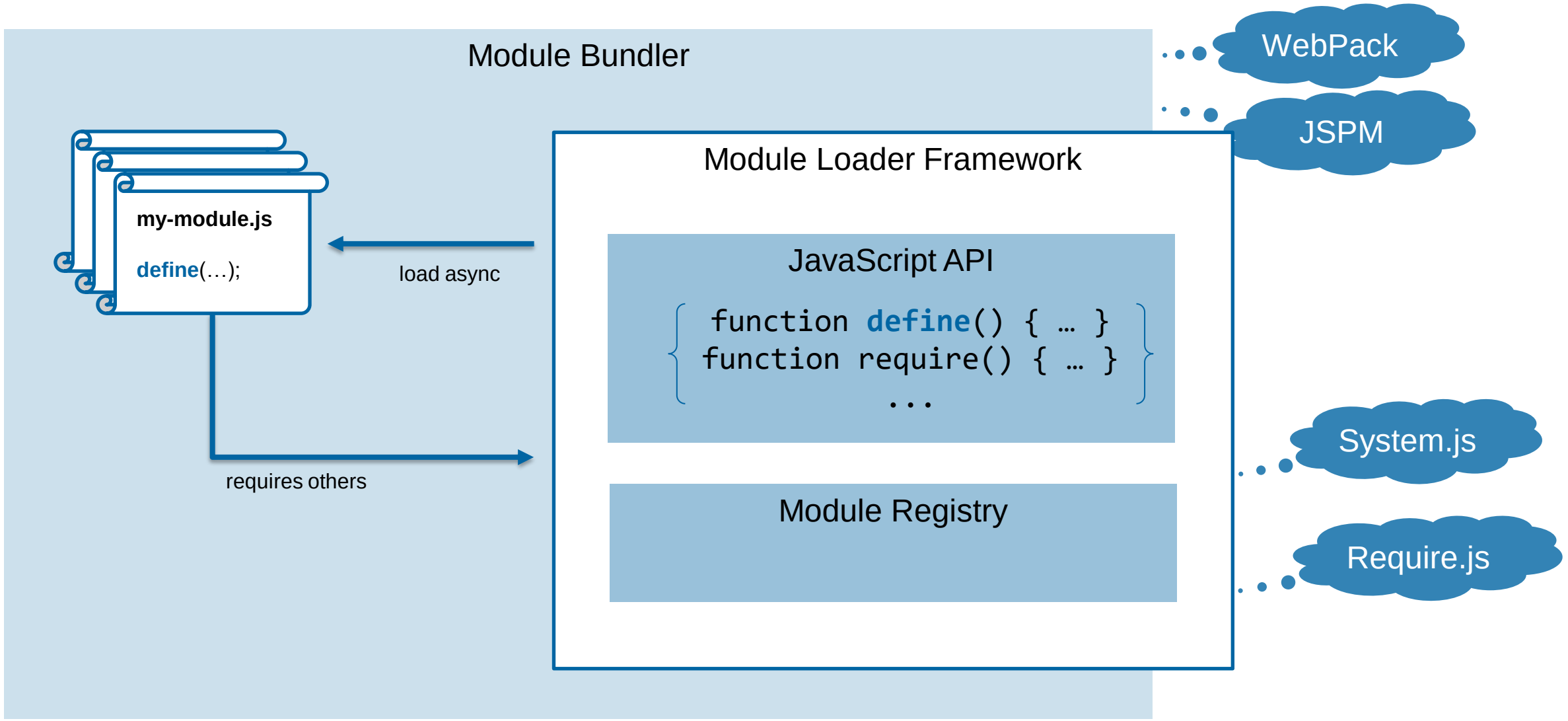
Asynchronous Module Definition (AMD) II

- **Intent**
  - Register the factory function by calling define(), instead of immediately executing it
    - Allows to load the dependent module asynchronously
  - Pass dependencies as an array of string values, do not grab globals directly
  - Name of the module is identified by its file name
    - Can be overwritten with named modules

- **Remarks**
  - For more about AMD see http://addyosmani.com/writing-modular-js/

- **Module Syntax Overview**
  - https://github.com/tiagorg/js-modules

Asynchronous Module Definition (AMD) Example



ES2015 Modules (ESM) I

- **ES2015 Modules have been influenced by CommonJS standard**
- ***strict mode* is turned on by default**
- **Modules are `import` / `export` statement based (new syntax)**
  - Uses `export` to expose variables/functions/classes to other modules
  - … and `import` to require variables/functions/classes from another module
- **Modules behave like `defer` annotated script-tags by default**
  - This prevents blocking the HTML parser while it fetches the module
    - …and it delays the script execution
  - Deferred scripts are executed in the order they're declared and before firing DOMContentLoaded event
  - *Remarks:* Inline module scripts are also deferred
- **Modules only execute once**
- **Each js-file defines its own module**

Source: https://jakearchibald.com/2017/es-modules-in-browsers/

ES2015 Modules (ESM) II

- **Due its different runtime behavior, scripts using the ESM syntax must be declared as “module”**
  - Add `type=module` on the inline/external script element

```
<script type="module">

    import {addTextToBody} from './utils.js';
    addTextToBody('Hello World!');

</script>
<script type="module" src="./main.js"> </script>
```

- **To hide scripts from browsers that understands modules, set `nomodule` attribute**
  - This way you can add backward compatibility for older browsers

```
<script type="module" src="./main.js"> </script>
<script nomodule src="./main_legacy.js"> </script>
```

Source: https://jakearchibald.com/2017/es-modules-in-browsers/

ES2015 Modules (ESM) Exports

- **To expose features of the current module, use `export` statement**
  - what to export and make available for other scripts (variable/function/class/…)
  - *Remarks:* ES2015 modules export bindings, not values or references
- **Examples**
  - `export let foo = 'bar';`
  - ```
    let foo = 'ponyfoo';
    let bar = 'baz';
    export { foo, bar };
    ```
  - `export default { foo, bar };`

```
// features in my-module.js …
// …

// public api
export default {
  foo: 'baz' // …
};
```

Source: Nicolás Bevacqua https://ponyfoo.com/articles/es6-modules-in-depth#importing-default-exports
```

ES2015 Modules (ESM) Import

- **To import features of another module, use `import` statement, followed by**
  - what to import and make available for your script (destructuring syntax, variable/function/class/…)
  - form where to import (module specifier)
- **Examples**
  - `import { bar } from 'https://jakearchibald.com/utils/bar.js';`
  - `import { foo, bar } from '/utils/bar.js';`
  - `import { bar } from './bar.js';`
  - *Remarks*: The imported name/s (bar) must be unique
- **Imports can also be renamed by using the `as` keyword**
  - `import { bar as utils_bar } from '/utils/bar.js';`
  - `import { bar as parent_bar } from '../bar.js';`
  - `import { * as all_my_globals } from '../all_globals.js';`
  - `import { default as default_exports } from '../default-exports.js';`
  - `import default_exports from '../default-exports.js';`

ES2015 Module versus AMD

ESM

```
import {dep1} from './src/dep1.js';
import {dep2} from './src/dep2.js';

// your JavaScript code
function doSomething() { }

export default doSomething;
```

my-module.js

AMD

```
define(
    'my-module',
    ['dep1', 'dep2'],
    function (dep1, dep2) {

    // your JavaScript code
    function doSomething() { }

    return {
        doSomething: doSomething
    };
});
```

my-module.js

Source: https://ponyfoo.com/articles/es6-modules-in-depth#importing-default-exports

Exercise 3 – Modularisiertes JavaScript

30min

Arbeiten Sie weiter an Ihrer Lösung aus Übung 2.

- **Lernziele**

  Klassen in JavaScript Module platzieren und mittels Import-/Export-Syntax referenzieren.

- **Lösen Sie `Exercise 3` (siehe Aufgaben im index.html File).**
- **Führen Sie nun ES2015 Modules ein.**
- **Falls Sie vorzeitig mit den Übungen fertig sind, lösen Sie die Zusatzaufgaben ganz unten im Abschnitt `Additional Exercise`.**

QUESTIONS?

Sources

- **Script**
  - Nirosh L.W.C. http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep

- **Slides**
  - http://www.jspatterns.com/category/patterns/object-creation/
  - http://addyosmani.com/resources/essentialjsdesignpatterns/book/
  - http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf
  - https://developer.mozilla.org/de/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript
  - https://packagecontrol.io/packages/JavaScript%20Patterns
  - http://en.wikipedia.org/wiki/Programming_idiom
  - http://javascript.crockford.com/